IN THE U.S. PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Applicant: TOSHIAKI YASUE et al.	:	Group Art Unit: 2192
Serial No.: 09/708,159	:	Examiner: J. Derek Rutten
Filed: November 8, 2000	:	February 12, 2007
Confirmation No.: 1032	:	William A. Kinnaman, Jr.
Title: PROGRAM EXECUTION METHOD	:	International Business Machines Corporation
	:	2455 South Road, Mail Station P386
	:	Poughkeepsie, NY 12601

APPLICANTS' APPEAL BRIEF

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

Applicants hereby submit their appeal brief in the above-identified application.

REAL PARTY IN INTEREST

The real party in interest is International Business Machines Corporation, the assignee of record.

RELATED APPEALS AND INTERFERENCES

There are no related appeals or interferences.

STATUS OF CLAIMS

Claims 1-10, constituting all of the currently pending claims, stand rejected and are being appealed. No claims have been allowed, nor have any claims been cancelled or withdrawn.

STATUS OF AMENDMENTS

There have been no amendments filed subsequent to final rejection.

SUMMARY OF CLAIMED SUBJECT MATTER

Claim 1

Claim 1 is directed to a program execution method for transferring, from an interpreter process (such as executed by the Java interpreter 124 of Fig. 1) to a loop process of a compiled code process (such as compiled by the JIT compiler 126 and executed by the native code execution unit 128 of Fig. 1), a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to the loop process of the compiled code process (page 8, lines 2-4).

In accordance with the invention, the loop process is first optimized. In this optimizing step, the one or more transfer points are moved to the top of the loop process if they can be moved there without a problem occurring (page 7, lines 15-16; page 10, lines 17-24). Also, code is copied from the top of the loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location immediately preceding the loop process if the transfer points are located inside the loop process (page 7, lines 18-21; page 10, line 26 to page 11, line 4).

Additionally, information is stored for generating recalculation code for one or more specific transfer points when privatization, common sub-expression elimination, and moving of code that

are performed pass beyond the specific transfer points (page 7, lines 24-26; page 11, lines 6-8), and a recalculation is performed during a transfer process (page 7, lines 26-27).

After the loop process has been optimized, execution is transferred from the interpreter process to the optimized loop process via one of the transfer points (Figs. 2-3, step S18; page 20, lines 22-23; page 21, lines 4-17).

Claim 4 is similar to claim 1, but is directed to a program storage device.

Claims 5 and 8

Claim 5 is directed to a program execution method for transferring, from an interpreter process (such as executed by the Java interpreter 124 of Fig. 1) to a loop process of a compiled code process (such as compiled by the JIT compiler 126 and executed by the native code execution unit 128 of Fig. 1), a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to the loop process of the compiled code process (page 8, lines 2-4).

In accordance with the invention, the loop process is first optimized by moving the one or more transfer points to the top of the loop process if they can be moved there without a problem occurring (page 7, lines 15-16; page 10, lines 17-24). After the loop process has been optimized, execution is transferred from the interpreter process to the optimized loop process via one of the transfer points (Figs. 2-3, step S18; page 20, lines 22-23; page 21, lines 4-17).

Claim 8 is similar to claim 5, but is directed to a program storage device.

Claims 7 and 10

Claim 7 is directed to a program execution method for transferring, from an interpreter process (such as executed by the Java interpreter 124 of Fig. 1) to a loop process of a compiled code process (such as compiled by the JIT compiler 126 and executed by the native code execution

unit 128 of Fig. 1), a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to the loop process of the compiled code process (page 8, lines 2-4).

In accordance with the invention, the loop process is first optimized by copying code from the top of the loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location immediately preceding the loop process if the transfer points are located inside the loop process (page 7, lines 18-21; page 10, line 26 to page 11, line 4). After the loop process has been optimized, execution is transferred from the interpreter process to the optimized loop process via one of the transfer points (Figs. 2-3, step S18; page 20, lines 22-23; page 21, lines 4-17).

Claim 10 is similar to claim 7, but is directed to a program storage device.

GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Claims 1-4 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Aho et al., Compilers: Principles, Techniques, and Tools, Chapter 10, "Code Optimization", pages 585-722 (1986) ("Aho") in view of Bak et al., U.S. Patent 6,513,156 ("Bak"), Bacon et al., "Compiler Transformations for High-Performance Computing", ACM Computing Surveys, vol. 26, no. 4, pages 345-420 (Dec. 1994) ("Bacon") and Koblenz et al., U.S. Patent 5,530,866, ("Koblenz").

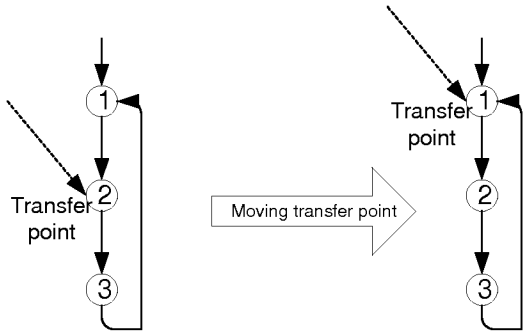
2. Claims 5-6 and 8-9 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Bak in view of Koblenz and Aho.

3. Claims 7 and 10 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Aho in view of Bak.

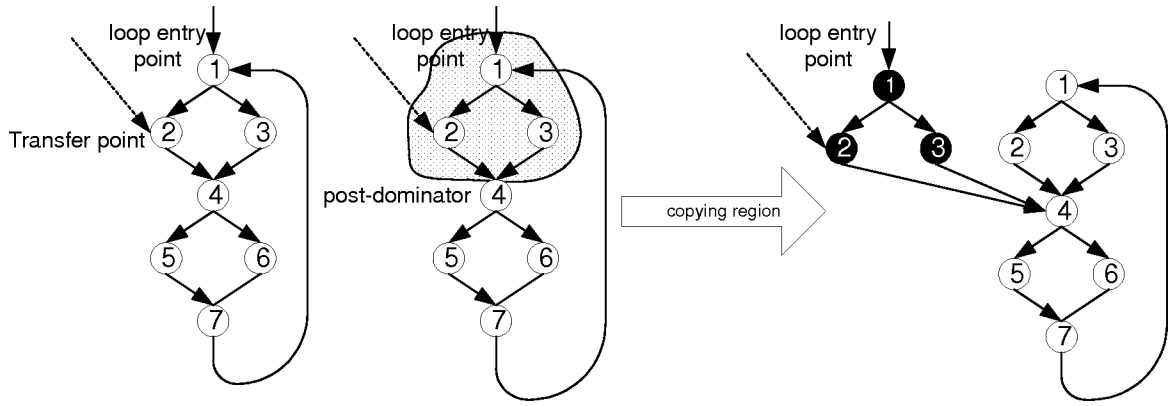
ARGUMENT

Introduction

Each of the claims on appeal recites an optimizing step including one or both of the steps of: (1) moving one or more transfer points to the top of a loop process if they can be moved there without a problem occurring; and (2) copying code from the top of the loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location immediately preceding the loop process if the transfer points are located inside the loop process. These steps are explained in more detail with reference to examples (a) and (b) below.



(a) An example of moving transfer point



(b) An example of copying a region from entry point to the post-dominating point

Example (a) above illustrates the moving step (1). Example (a) shows on the left a loop (nodes 1-3) having an entry point (node 1) at its top as well as a transfer point (node 2) along its length to which control may pass other than through the entry point (i.e., from another routine). Because of the transfer point at node 2, the loop is considered to be "irreducible" and many common optimization techniques cannot be used. In accordance with this aspect of the invention, the transfer point is moved (if this can be done without a problem occurring) from node 2 to node 1 at the top of the loop, as shown on the right in the figure, so that the loop is now reducible.

Example (b) above illustrates the copying step (2). Example (b) shows a loop (nodes 1-7) having an entry point (node 1) and a transfer point (node 2) to which control may pass other than through the entry point. Node 4 of the loop is said to "post-dominate" nodes 1 and 2, since every path of execution passing through node 1 or 2 subsequently passes through node 4. Again, because of the transfer point at node 2, the loop is irreducible and many common optimization techniques cannot be used.

In accordance with this aspect of the invention, the region (nodes 1-3) from the top of the loop (node 1) to a point (node 4) that post-dominates the top of the loop and the transfer point (node 2) is copied to a location immediately preceding the loop. More particularly, nodes 1-3 are copied as nodes 1′-3′ (as the nodes shown as black circles will be identified here), with node 1′ having directed edges to nodes 2′ and 3′ and with nodes 2′ and 3′ having directed edges to post-dominating node 4. With this change, the loop containing nodes 1-7 has but a single entry point (now node 4), allowing conventional optimization techniques to be used.

These techniques are used to "reduce" a loop of a compiled code process, such as one generated by a just-in-time (JIT) Java bytecode compiler, containing transfer points at which execution is transferred from an interpreter process such as that of a Java bytecode interpreter. Reducing the loop in such a manner permits the use of loop optimization techniques that are not otherwise usable.

Claims 1 and 4

Claim 1, representative of this group of claims on appeal, reads as follows (with paragraph numbers added for later reference):

> 1. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:
>
> [a] optimizing the loop process, said optimizing step including the steps of:
>
> [a1] moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring;
>
> [a2] copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process;
>
> [a3] storing information for generating recalculation code for one or more specific transfer points when privatization, common sub-expression elimination, and moving of code that are performed pass beyond said specific transfer points; and
>
> [a4] performing a recalculation during a transfer process; and
>
> [b] transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

As already noted above, claim 4 is similar to claim 1 but is directed to a program storage device.

Both claim 1 and claim 4 recite the step (a1) of moving the one or more transfer points to the top of the loop process if they can be moved there without a problem occurring (example (a) above), as well as the step (a2) of copying code from the top of the loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location

immediately preceding the loop process if the transfer points are located inside the loop process (example (b) above).

Claims 1-4[1] stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Aho et al., Compilers: Principles, Techniques, and Tools, Chapter 10, "Code Optimization", pages 585-722 (1986) ("Aho") in view of Bak et al., U.S. Patent 6,513,156 ("Bak"), Bacon et al., "Compiler Transformations for High-Performance Computing", ACM Computing Surveys, vol. 26, no. 4, pages 345-420 (Dec. 1994) ("Bacon") and Koblenz et al., U.S. Patent 5,530,866, ("Koblenz") (Final Action mailed Aug. 11, 2006,[2] pages 4-9, ¶ 9). These references, and their use in rejecting claims 1 and 4, are next discussed.

Aho discusses code optimization, including loop optimization in particular. In making the above rejection, the Examiner cites Aho (page 5) for its alleged teaching of several aspects of the overall combination, including the code copying step (a2). The Examiner concedes, however (pages 5-6), that Aho does not "expressly disclose": (1) moving transfer points to the top of a loop process (step a1); (2) transferring a method from an interpreter process to a compiled code process (step 2); (3) storing information for generating recalculation code for specific transfer points (step a3); (4) performing a recalculation during a transfer process (step a4); and (5) privatization (in step a3).

Bak discloses a Java virtual machine (JVM) that alternates (as does applicants' system) between interpretive execution of Java bytecode by a Java interpreter and native execution of Java bytecode that has been compiled to run on the underlying CPU architecture (Fig. 5). The Examiner cites Bak (pages 6-7) for its alleged teaching of transferring a method from an interpreter process to a compiled code process (step 2); storing information for generating recalculation code for specific transfer points (step a3); and performing a recalculation during a transfer process (step a4).

[1] The actual statement of rejection on page 4 refers to claims 1-10. However, the cited references are only applied against claims 1-4, and claims 5-10 are the subjects of separate obviousness rejections on subcombinations of these references.

Bacon relates to compiler transformations, as its title suggests, and is cited (page 7) for its alleged teaching (beginning at page 395, § 7.1.3) of privatization.

Koblenz relates to "methods for allocating physical registers within a compiler phase to achieve efficient operation of a target CPU" (abstract) and is cited (page 6) for its alleged teaching (at col. 8, lines 23-28) of the step (a1) of moving one or more transfer points to the top of a loop process if they can be moved there without a problem occurring.

In rejecting claims 1 and 4, the Examiner argues that it would have been obvious "to use Koblenz' moving of transfer points with Bacon's optimizations with Bak's mixed mode interpreter in Aho' code optimizer" (page 7). As the Examiner explains (Id.):

> One of ordinary skill would have been motivated to remove transfer points from a loop in order to make them reducible since many optimizations depend on reducibility (Aho page 607). Further, one would have been motivated to transfer the execution of an interpreted loop to natively compiled instructions since native code executes more quickly than interpreted code.

For the purposes of this appeal, applicants do not contest the pertinence of Bak and Bacon as references, nor their combination with Aho in the manner suggested by the Examiner. However, applicants do contest two of the premises underlying the Examiner's rejection. The first is Koblenz's alleged teaching of moving one or more transfer points to the top of a loop process if they can be moved there without a problem occurring. The second is Aho's alleged teaching of copying code from the top of a loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location immediately preceding the loop process if the transfer points are located inside the loop process. These will be discussed in turn.

---

[2] Subsequent references herein to the Examiner's actions or statements are to this Office communication unless otherwise indicated.

1. Moving Transfer Points to Top of Loop Process

This aspect of applicants' invention, shown in example (a) above and recited in step a1 of claims 1 and 4, involves moving one or more transfer points, at which program execution is transferred from an interpreter process to a loop process of a compiled code process, to the top of the loop process if they can be moved there without a problem occurring.

In addressing this feature, the Examiner relies on Koblenz's supposed teaching of the movement of transfer points to the top of a loop process. As the abstract states, Koblenz relates to methods for "allocating physical registers within a compiler phase to achieve efficient operation of a target CPU." The operation of this system involves in part the analysis of a "tile tree" (Fig. 1b), defined as "[a] structure of tiles representing the loop and conditional structure of the code under analysis". As the patentee elaborates at column 8, lines 15-28 (emphasis added):

> This loop structure of the code is identified on the basis of intervals in the control flow graph as accomplished in prior art processes. An interval in the control flow graph is a set of basic blocks that form a loop in the code. Intervals, like tiles and loops, nest.
>
> In a tile tree construction process useful in the practice of the present invention, a "loop top" is defined as the single basic block having incoming back edges that dominates every basic block in its loop. Irreducible loops do not exhibit a loop top; however, all basic blocks in an irreducible loop that are reached by a forward control flow edge from a basic block outside the loop can be combined into a single summary loop top in constructing the tile tree. This summary node will dominate every basic block in the loop.

The Examiner argues that the highlighted language teaches moving transfer points to the top of a loop process, as recited in step a1 of claims 1 and 4. However, nothing of this sort is evident from the highlighted language. Rather, what this passage appears to suggest is that, when constructing a tile tree, one can combine all basic blocks in an irreducible loop that are reached by a forward control flow edge from a basic block outside the loop into a single "summary loop top". Whatever utility this may have for constructing a tile tree, it does not appear to address the

problem of loop reducibility by converting an irreducible loop to a reducible one. It merely accepts the irreducibility of the loop and creates a composite "summary loop top" for the limited purpose of constructing a tile tree.

Moreover, even if Koblenz's formation of a single summary loop top could be equated with applicants' claimed moving step (a1) (which it cannot for the reasons given above), it is merely the manipulation of an abstract flow graph for the purpose of assigning physical registers, not the modification of a loop process for the purpose of code optimization, as claimed by applicants. Accordingly, no one interested in modifying a loop process for the latter purpose would even consult this reference in the first place.

The Examiner concedes that Koblenz relates to manipulation of an abstract flow graph, but argues that such manipulation "is helpful in loop optimization" (page 4). The Examiner argues further that, regardless of Koblenz's use of loop manipulation for physical register assignment, the teaching of loop manipulation in an abstract flow graph "provides valuable information regarding the nature of domination and irreducibility" (Id.). Finally, the Examiner argues that Koblenz is not relied upon for loop optimization and that Aho is instead relied on for this purpose (Id.).

Undoubtedly, manipulation of an abstract flow graph may be "helpful for loop optimization" and may provide "valuable information regarding the nature of domination and irreducibility", as the Examiner suggests, but only if done so for a comparable purpose and with a comparable result. Such is not the case here, where the purpose is register allocation and the result is a composite "summary loop top" rather than an actual loop top that can be traversed by code. Further, while the Examiner may rely on Aho for the general notion of loop optimization, for Koblenz to be used there must be something in that reference itself suggesting its relevance to the problems of loop optimization and reducibility. As noted above, Koblenz does not solve the problem of loop irreducibility, but merely sidesteps the issue through the device of a "summary loop top" for the purpose of tile construction.

Accordingly, Koblenz does not teach moving one or more transfer points at which program execution is transferred to a loop process to the top of the loop process, as recited in claims 1 and 4. Therefore, the Examiner's rejection of these claims on a combination of references including this patent is untenable and should be reversed.

2. Copying Code from Top of Loop Process

This aspect of applicants' invention, shown in example (b) above and recited in step a2 of claims 1 and 4, involves copying code, extending from the top of the loop process to a point that post-dominates the top of the loop process and the transfer points, to a location immediately preceding the loop process if the transfer points are located inside the loop process.

The Examiner points to several excerpts from Aho, the only reference cited on this aspect of code copying. None of these excerpts, however, teach this aspect of applicants' claimed invention. The Examiner first points to the discussion of various code optimizations at pages 664-668, in particular the discussion of node splitting beginning at the bottom of page 666 (page 5). More relevantly, the Examiner points to the node-splitting example, described as Example 10.36 on page 667 and shown in Fig. 10.49 on page 668. There, the node 2 of Fig. 10.49(a) is split into the pair of nodes 2a and 2b shown in Fig. 10.49(b), permitting the further simplifications shown in Fig. 10.49(c) and (d). However, while this shows copying of a node, there are no depicted entry points or transfer points to the flow graph formed by the nodes illustrated in Fig. 10.49(a). Thus, this portion of Aho does not teach copying code represented by a set of (one or more) nodes, extending from the top of a loop to a point that post-dominates the top of the loop and a transfer point, to a location immediately preceding the loop as claimed by applicants.

The same comments apply to the other node-splitting example cited, shown in Fig. 10.57 on page 680 and described on pages 679-680. There, the node on the left, containing a pair of dots, is split into a pair of such nodes, in the same manner as node 2 in Fig. 10.49. Again, this only shows node splitting in the abstract, without any relation to entry points or transfer points to a

loop. It does not teach copying code represented by a set of such nodes, extending from the top of a loop to a point that post-dominates the top of the loop and a transfer point.

Accordingly, Aho fails to teach copying code, extending from the top of a loop process to a point that post-dominates the top of the loop process and one or more transfer points, to a location immediately preceding the loop process, as recited in claims 1 and 4. Therefore, the Examiner's rejection of these claims on the asserted combination of references is untenable and should be reversed on this ground as well.

Claims 5 and 8

Claim 5, representative of this group of claims on appeal, reads as follows:

> 5. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:
>
> optimizing the loop process, said optimizing step including the step of moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring; and
>
> transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

As already noted above, claim 8 is similar to claim 5 but is directed to a program storage device. Like claim 5, claim 8 recites the step of moving the one or more transfer points to the top of the loop process if they can be moved there without a problem occurring.

Claims 5 and 8 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Bak in view of Koblenz and Aho (pages 9-10, ¶ 10). In making this rejection, the Examiner basically reiterates his contentions regarding claims 1 and 4. More particularly, the Examiner contends

that it would have been obvious "to use Koblenz' movement of transfer points, and Aho's optimization with Bak's transfer of execution", since one of ordinary skill "would have been motivated to make a loop reducible in order to better optimize it" (page 9).

This rejection uses Koblenz in the same manner as in the rejection of claims 1 and 4 and fails for the same reason. That is, even assuming that it would be obvious to make a loop reducible in order to better optimize it, Koblenz does not teach that this can be achieved by moving transfer points. Accordingly, the Examiner's rejection of claims 5 and 8 on this combination of references is likewise untenable and should be reversed.

Claims 6 and 9

Claim 6, representative of this group of claims on appeal, is dependent on claim 5 and reads as follows:

> 6. The program execution method according to claim 5, said optimizing step further including the step of:
>
> copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process.

Claim 9 is similar to claim 6 but is directed to a program storage device. Like claim 6, claim 9 recites not only the step of moving the one or more transfer points to the top of the loop process, but also the step of copying code from the top of the loop process to a point that post-dominates the top of the loop process and the one or more transfer points to a location immediately preceding the loop process.

Claims 6 and 9, like claims 5 and 8, stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Bak in view of Koblenz and Aho (pages 9-10, ¶ 10). With regard to the additional limitation of the copying step, the Examiner again draws on his contentions against claim 1, arguing that it would have been obvious "to use Aho's code copying with Bak's transfer

process in order to facilitate the reducibility of a graph which would allow better optimization" (page 9).

This rejection thus uses Aho in the same manner as in the rejection of claims 1 and 4 and fails for the same reason. That is, even assuming that it would be obvious to make a loop reducible in order to better optimize it, Aho does not teach that this can be achieved by copying code in the manner claimed. Accordingly, the Examiner's rejection of claims 6 and 9 on this combination of references is untenable for this reason as well, in addition to the reasons applicable to the base claims 5 and 8, and should be reversed.

Claims 7 and 10

Claim 7, representative of this group of claims on appeal, reads as follows:

> 7. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:
>
> optimizing the loop process, said optimizing step including the step of copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process; and
>
> transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

As noted above, claim 10 is similar to claim 7 but is directed to a program storage device. Both of these claims thus recite the step of copying code from the top of a loop process to a point that post-dominates the top of the loop process and one or more transfer points to a location immediately preceding the loop process if the transfer points are located inside the loop process.

Claims 7 and 10[3] stand rejected under 35 U.S.C. § 103(a) as being unpatentable over Aho in view of Bak, with the references being applied as against claims 1-4 (page 10, ¶ 11). Since, for the reasons given above, Aho does not teach the claimed copying step, this rejection is untenable and should be reversed.

Conclusion

For the foregoing reasons, the Examiner's rejection of claims 1-10 on the art cited is untenable and should be reversed.

Respectfully submitted,
TOSHIAKI YASUE et al.

By /William A. Kinnaman, Jr./
William A. Kinnaman, Jr., Reg. No. 27,650
Voice: (845) 433-1175
Fax: (845) 432-9601

WAK/wak

[3] The actual statement of rejection on page 10 refers only to claim 7. However, the cited references are applied against claims 7 and 10.

CLAIMS APPENDIX

1. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:

optimizing the loop process, said optimizing step including the steps of:

moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring;

copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process;

storing information for generating recalculation code for one or more specific transfer points when privatization, common sub-expression elimination, and moving of code that are performed pass beyond said specific transfer points; and

performing a recalculation during a transfer process; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

2. The program execution method according to claim 1, further comprising the step of:

defining as a new transfer point, a point from said interpreter process to said compiled code process whereat, when said method that is currently being executed is replaced, the execution speed is increased compared with when said method is not replaced.

3. The program execution method according to claim 1 or 2, further comprising the steps of:

generating information required to perform a transfer from said interpreter process to said compiled code process; and

storing said generated information while correlating said generated information with said transfer points,

wherein, at said recalculation step, said information stored for said transfer points is employed.

4. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, the method steps comprising:

optimizing the loop process, said optimizing step including the steps of:

moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring;

copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process;

storing information for generating recalculation code for one or more specific transfer points when privatization, common sub-expression elimination, and moving of code that are performed pass beyond said specific transfer points; and

performing a recalculation during a transfer process; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

5. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:

optimizing the loop process, said optimizing step including the step of moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

6. The program execution method according to claim 5, said optimizing step further including the step of:

copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process.

7. A program execution method for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, comprising the steps of:

optimizing the loop process, said optimizing step including the step of copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

8. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, the method steps comprising:

optimizing the loop process, said optimizing step including the step of moving said one or more transfer points to the top of said loop process if they can be moved there without a problem occurring; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

9. The program storage device of claim 8, said optimizing step further including the step of:

copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process.

10. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for transferring, from an interpreter process to a loop process of a compiled code process, a method that is currently being executed for code that includes one or more transfer points at which program execution is transferred from the interpreter process to said loop process of the compiled code process, the method steps comprising:

optimizing the loop process, said optimizing step including the step of copying code from the top of the loop process to a point that post-dominates said top of said loop process and said one or more transfer points to a location immediately preceding said loop process if said transfer points are located inside said loop process; and

transferring execution from the interpreter process to the optimized loop process via one of said transfer points.

EVIDENCE APPENDIX

(None)

RELATED PROCEEDINGS APPENDIX

(None)